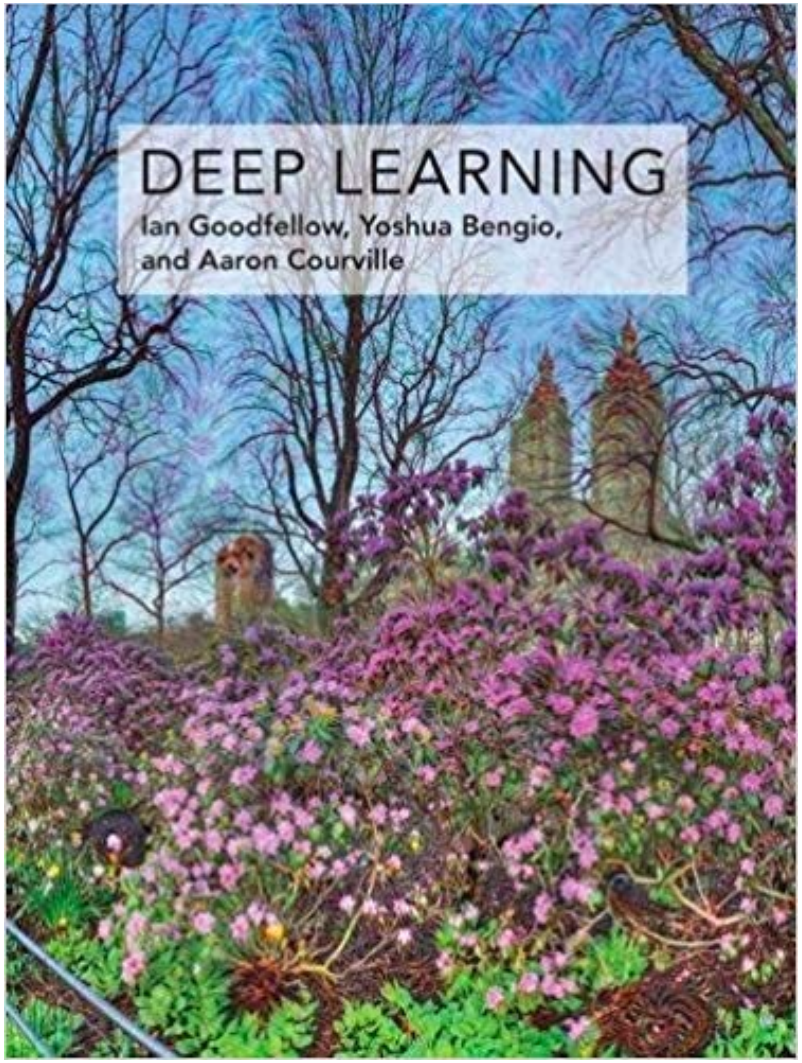
國立陽明交通大學
NATIONAL YANG MING CHIAO TUNG UNIVERSITY

Deep Learning
深度學習
Fall 2025

Back Propagation
(Chapter 6.5)

Prof. Chia-Han Lee
李佳翰 教授



- Figure source: Textbook and Internet
- You are encouraged to buy the textbook.
- Please respect the copyright of the textbook. Do not distribute the materials to other people.

Back-propagation

- When we use a feedforward neural network to accept an input x and produce an output $\hat{\boldsymbol{y}}$, information flows forward through the network. The input $\boldsymbol{x}$ provides the initial information that then propagates up to the hidden units at each layer and finally produces $\hat{\boldsymbol{y}}$. This is called forward propagation.

- During training, forward propagation can continue onward until it produces a scalar cost $J(\boldsymbol{\theta})$.

- The back-propagation algorithm, often simply called backprop, allows the information from the cost to then flow backward through the network in order to compute the gradient.

Back-propagation

- Computing an analytical expression for the gradient is straightforward, but numerically evaluating such an expression can be computationally expensive. The back-propagation algorithm uses a simple and inexpensive procedure to calculate the gradient.

- The term back-propagation is often misunderstood as meaning the whole learning algorithm. Actually, back-propagation refers only to the method for computing the gradient, while (stochastic) gradient descent algorithm is used to perform learning using this gradient.

- Back-propagation is often misunderstood as being specific to multi-layer neural networks, but in principle it can compute derivatives of any function.

Back-propagation

- Specifically, we will describe how to compute the gradient $\nabla_{\boldsymbol{x}} f(\boldsymbol{x}, \boldsymbol{y})$ for an arbitrary function f, where $\boldsymbol{x}$ is a set of variables whose derivatives are desired, and $\boldsymbol{y}$ is an additional set of variables that are inputs to the function but whose derivatives are not required.

- In learning algorithms, the gradient we most often require is the gradient of the cost function with respect to the parameters, $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$.

Computational graphs

- Many ways of formalizing computation as graphs are possible. Here, we use each node in the graph to indicate a variable. The variable may be a scalar, vector, matrix, tensor, or even a variable of another type.

- An operation is a simple function of one or more variables. Functions more complicated than the operations in this set may be described by composing many operations together.

Computational graphs

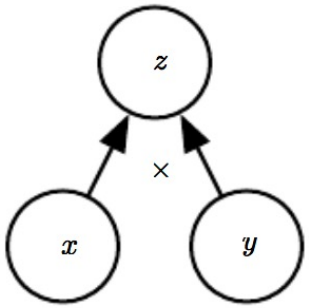
- Without loss of generality, we define an operation to return only a single output variable. This does not lose generality because the output variable can have multiple entries, such as a vector.

- Software implementations of back-propagation usually support operations with multiple outputs, but we avoid this case because it introduces many extra details that are not important to conceptual understanding.

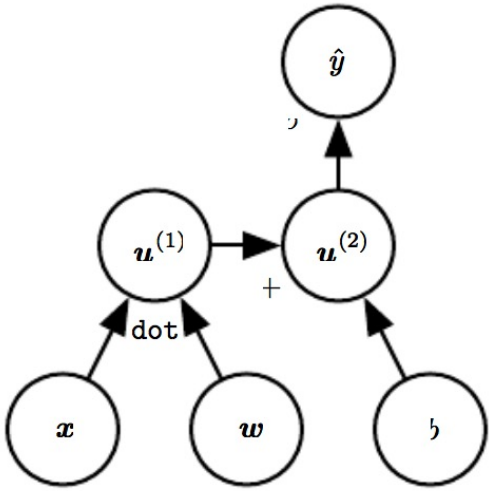
- If a variable y is computed by applying an operation to a variable x, then we draw a directed edge from x to y.

- We sometimes annotate the output node with the name of the operation applied, and other times omit this label when the operation is clear from context.

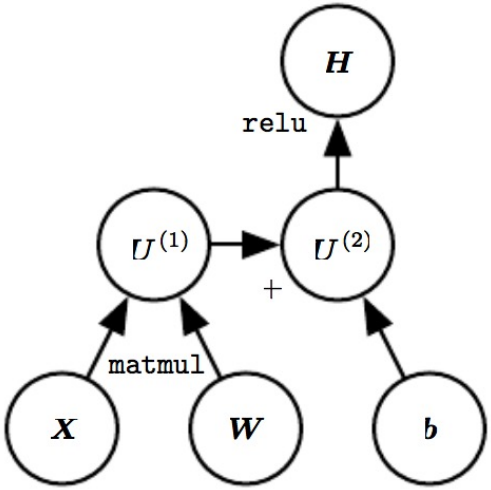
Computational graphs



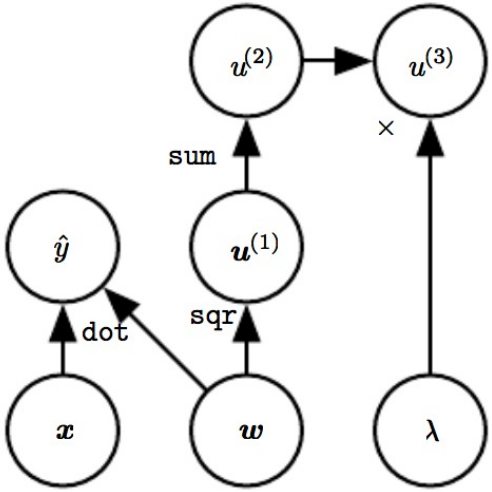
(a)



(b)



(c)



(d)

Chain rule of calculous

- The chain rule of calculus is used to compute the derivatives of functions formed by composing other functions whose derivatives are known.

- Back-propagation is an algorithm that computes the chain rule, with a specific order of operations that is highly efficient.

- Let x be a real number, and let f and g both be functions mapping from a real number to a real number. Suppose that $y = g(x)$ and $z = f(g(x)) = f(y)$. Then the chain rule states that

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}. \tag{6.44}$$

Chain rule of calculous

- We can generalize this beyond the scalar case. Suppose that $\boldsymbol{x} \in \mathbb{R}^m$, $\boldsymbol{y} \in \mathbb{R}^n$, g maps from $\mathbb{R}^m$ to $\mathbb{R}^n$, and f maps from $\mathbb{R}^n$ to $\mathbb{R}$. If $\boldsymbol{y} = g(\boldsymbol{x})$ and $z = f(\boldsymbol{y})$, then

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}. \tag{6.45}$$

- In vector notation, this may be equivalently written as

$$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^{\top} \nabla_{\boldsymbol{y}} z, \tag{6.46}$$

where $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ is the $n \times m$ Jacobian matrix of g. The gradient of a variable $\boldsymbol{x}$ can be obtained by multiplying a Jacobian matrix $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ by a gradient $\nabla_{\boldsymbol{y}} z$. The back-propagation algorithm consists of performing such a Jacobian-gradient product for each operation in the graph.

Chain rule of calculous

- Usually we apply the back-propagation algorithm to tensors of arbitrary dimensionality, not merely to vectors. Conceptually, this is exactly the same as back-propagation with vectors. The only difference is how the numbers are arranged in a grid to form a tensor.

- We could imagine flattening each tensor into a vector before we run back-propagation, computing a vector-valued gradient, and then reshaping the gradient back into a tensor. In this rearranged view, back-propagation is still just multiplying Jacobians by gradients.

Chain rule of calculous

- To denote the gradient of a value z with respect to a tensor $\mathbf{X}$, we write $\nabla_{\mathbf{X}} z$, just as if $\mathbf{X}$ were a vector. The indices into $\mathbf{X}$ now have multiple coordinates—for example, a 3-D tensor is indexed by three coordinates.

- We can abstract this away by using a single variable i to represent the complete tuple of indices. For all possible index tuples i, $(\nabla_{\mathbf{X}} z)_i$ gives $\frac{\partial z}{\partial \mathbf{X}_i}$. This is exactly the same as how for all possible integer indices i into a vector, $(\nabla_{\boldsymbol{x}} z)_i$ gives $\frac{\partial z}{\partial x_i}$. Using this notation, we can write the chain rule as it applies to tensors. If $\mathbf{Y} = g(\mathbf{X})$ and $z = f(\mathbf{Y})$, then

$$\nabla_{\mathbf{X}} z = \sum_j (\nabla_{\mathbf{X}} Y_j) \frac{\partial z}{\partial Y_j}. \tag{6.47}$$

Recursively applying the chain rule

- Using the chain rule, it is straightforward to write down an algebraic expression for the gradient of a scalar with respect to any node in the computational graph that produced that scalar.

- Actually evaluating that expression in a computer, however, introduces some extra considerations. Specifically, many subexpressions may be repeated several times within the overall expression for the gradient. Any procedure that computes the gradient will need to choose whether to store these subexpressions or to recompute them several times.

Recursively applying the chain rule

- In some cases, computing the same subexpression twice would simply be wasteful. For complicated graphs, there can be exponentially many of these wasted computations, making a naive implementation of the chain rule infeasible.

- In other cases, computing the same subexpression twice could be a valid way to reduce memory consumption at the cost of higher runtime.

- We begin with a version of the back-propagation algorithm that specifies the actual gradient computation directly, in the order it will actually be done and according to the recursive application of chain rule.

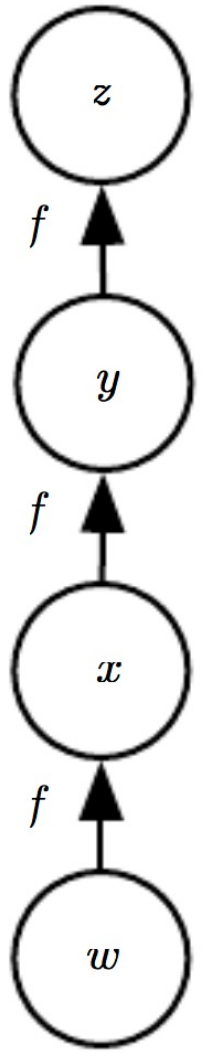
Recursively applying the chain rule

$$\frac{\partial z}{\partial w} \tag{6.49}$$

$$= \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w} \tag{6.50}$$

$$= f'(y)f'(x)f'(w) \tag{6.51}$$

$$= f'(f(f(w)))f'(f(w))f'(w). \tag{6.52}$$



- Equation 6.51 suggests an implementation in which we compute the value of $f(\omega)$ only once and store it in the variable x. This is the approach taken by the back-propagation algorithm.

- An alternative approach is suggested by equation 6.52, where the subexpression $f(\omega)$ appears more than once. In the alternative approach, $f(\omega)$ is recomputed each time it is needed.

Recursively applying the chain rule

- First consider a computational graph describing how to compute a single scalar $u^{(n)}$ (say, the loss on a training example). This scalar is the quantity whose gradient we want to obtain, with respect to the n_i input nodes $u^{(1)}$ to $u^{(n_i)}$. In other words, we wish to compute $\frac{\partial u^{(n)}}{\partial u^{(i)}}$ for all $i \in \{1, 2, \ldots, n_i\}$.

- In the application of back-propagation to computing gradients for gradient descent over parameters, $u^{(n)}$ will be the cost associated with an example or a minibatch, while $u^{(1)}$ to $u^{(n_i)}$ correspond to the parameters of the model.

Recursively applying the chain rule

- We will assume that the nodes of the graph have been ordered in such a way that we can compute their output one after the other, starting at $u^{(n_i+1)}$ and going up to $u^{(n)}$. As defined in algorithm 6.1, each node $u^{(i)}$ is associated with an operation $f^{(i)}$ and is computed by evaluating the function

$$u^{(i)} = f(\mathbb{A}^{(i)}), \tag{6.48}$$

where $\mathbb{A}^{(i)}$ is the set of all nodes that are parents of $u^{(i)}$.

Recursively applying the chain rule

- Algorithm 6.1 specifies the forward propagation computation, which we could put in a graph $\mathcal{G}$.

for $i = 1, \ldots, n_i$ **do**
  $u^{(i)} \leftarrow x_i$
end for
for $i = n_i + 1, \ldots, n$ **do**
  $\mathbb{A}^{(i)} \leftarrow \{u^{(j)} \mid j \in Pa(u^{(i)})\}$
  $u^{(i)} \leftarrow f^{(i)}(\mathbb{A}^{(i)})$
end for
return $u^{(n)}$

Recursively applying the chain rule

- To perform back-propagation, we construct a computational graph that depends on $\mathcal{G}$ and adds to it an extra set of nodes. These form a subgraph $\mathcal{B}$ with one node per node of $\mathcal{G}$.

- Computation in $\mathcal{B}$ proceeds in exactly the reverse of the order of computation in $\mathcal{G}$, and each node of $\mathcal{B}$ computes the derivative $\frac{\partial u^{(n)}}{\partial u^{(i)}}$ associated with the forward graph node $u^{(i)}$. This is done using the chain rule with respect to scalar output $u^{(n)}$:

$$\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j \in Pa(u^{(i)})} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}} \tag{6.53}$$

as specified by Algorithm 6.2.

Recursively applying the chain rule

Algorithm 6.2

Run forward propagation (algorithm 6.1 for this example) to obtain the activations of the network.

Initialize `grad_table`, a data structure that will store the derivatives that have been computed. The entry `grad_table`$[u^{(i)}]$ will store the computed value of $\frac{\partial u^{(n)}}{\partial u^{(i)}}$.

`grad_table`$[u^{(n)}] \leftarrow 1$

for $j = n - 1$ down to 1 **do**

The next line computes $\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j \in Pa(u^{(i)})} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}}$ using stored values:

`grad_table`$[u^{(j)}] \leftarrow \sum_{i:j \in Pa(u^{(i)})}$ `grad_table`$[u^{(i)}] \frac{\partial u^{(i)}}{\partial u^{(j)}}$

end for

return {`grad_table`$[u^{(i)}] \mid i = 1, \ldots, n_i$}

Recursively applying the chain rule

- The subgraph $\mathcal{B}$ contains exactly one edge for each edge from node $u^{(j)}$ to node $u^{(i)}$ of $\mathcal{G}$. The edge from $u^{(j)}$ to $u^{(i)}$ is associated with the computation of $\frac{\partial u^{(i)}}{\partial u^{(j)}}$.

- A dot product is performed for each node, between the gradient already computed with respect to nodes $u^{(i)}$ that are children of $u^{(j)}$ and the vector containing the partial derivatives $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ for the same children nodes $u^{(i)}$.

- To summarize, the amount of computation required for performing the back-propagation scales linearly with the number of edges in $\mathcal{G}$, where the computation for each edge corresponds to computing a partial derivative as well as performing one multiplication and one addition.

Back-propagation in fully connected MLP

- Let us consider the specific graph associated with a fully-connected multi layer MLP. Algorithm 6.3 first shows the forward propagation, which maps parameters to the supervised loss $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$ associated with a single (input,target) training example $(\boldsymbol{x}, \boldsymbol{y})$, with $\hat{\boldsymbol{y}}$, the output of the neural network when $\boldsymbol{x}$ is provided in input.

Require: Network depth, l
Require: $\boldsymbol{W}^{(i)}, i \in \{1, \ldots, l\}$, the weight matrices of the model
Require: $\boldsymbol{b}^{(i)}, i \in \{1, \ldots, l\}$, the bias parameters of the model
Require: $\boldsymbol{x}$, the input to process
Require: $\boldsymbol{y}$, the target output

$\boldsymbol{h}^{(0)} = \boldsymbol{x}$
for $k = 1, \ldots, l$ **do**
  $\boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)} \boldsymbol{h}^{(k-1)}$
  $\boldsymbol{h}^{(k)} = f(\boldsymbol{a}^{(k)})$
end for
$\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)}$
$J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda \Omega(\theta)$

Back-propagation in fully connected MLP

- Algorithm 6.4 then shows the corresponding computation to be done for applying the back-propagation algorithm to this graph.

After the forward computation, compute the gradient on the output layer:

$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$

for $k = l, l-1, \ldots, 1$ **do**

Convert the gradient on the layer's output into a gradient into the pre-nonlinearity activation (element-wise multiplication if f is element-wise):

$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$

Compute gradients on weights and biases (including the regularization term, where needed):

$\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\theta)$

$\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}\, \boldsymbol{h}^{(k-1)\top} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega(\theta)$

Propagate the gradients w.r.t. the next lower-level hidden layer's activations:

$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)\top} \boldsymbol{g}$

end for

Back-propagation in fully connected MLP

- Algorithms 6.3 and 6.4 are demonstrations chosen to be simple and straightforward to understand. However, they are specialized to one specific problem.

- Modern software implementations are based on the generalized form of back-propagation, which can accommodate any computational graph by explicitly manipulating a data structure for representing symbolic computation.

General back-propagation

- To compute the gradient of some scalar z with respect to one of its ancestors $\boldsymbol{x}$ in the graph, we begin by observing that the gradient with respect to z is given by $\frac{dz}{dz} = 1$.

- We can then compute the gradient with respect to each parent of z in the graph by multiplying the current gradient by the Jacobian of the operation that produced z.

- We continue multiplying by Jacobians, traveling backward through the graph in this way until we reach $\boldsymbol{x}$. For any node that may be reached by going backward from z through two or more paths, we simply sum the gradients arriving from different paths at that node.

General back-propagation

- Assume that each operation evaluation has roughly the same cost. Computing a gradient in a graph with n nodes will never execute more than $O(n^2)$ operations or store the output of more than $O(n^2)$ operations.

- Here we are counting operations in the computational graph, not individual operations executed by the underlying hardware, so the runtime of each operation may be highly variable. For example, multiplying two matrices that each contain millions of entries might correspond to a single operation in the graph.

- Computing the gradient requires at most $O(n^2)$ operations because the forward propagation stage will at worst execute all n nodes in the original graph.

General back-propagation

- The back-propagation algorithm adds one Jacobian-vector product, which should be expressed with $O(1)$ nodes, per edge in the original graph. Because the computational graph is a directed acyclic graph it has at most $O(n^2)$ edges.

- Most neural network cost functions are roughly chain-structured, causing back-propagation to have $O(n)$ cost.

General back-propagation

- The naive approach might need to execute exponentially many nodes. This potentially exponential cost can be seen by expanding and rewriting the recursive chain rule (equation 6.53) nonrecursively:

$$\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{\substack{\text{path } (u^{(\pi_1)}, u^{(\pi_2)}, \ldots, u^{(\pi_t)}), \\ \text{from } \pi_1 = j \text{ to } \pi_t = n}} \prod_{k=2}^{t} \frac{\partial u^{(\pi_k)}}{\partial u^{(\pi_{k-1})}}. \tag{6.55}$$

- Since the number of paths from node j to node n can grow exponentially in the length of these paths, the number of terms in the above sum, which is the number of such paths, can grow exponentially with the depth of the forward propagation graph.
- This large cost would be incurred because the same computation for $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ would be redone many times.

General back-propagation

- To avoid such recomputation, we can think of back-propagation as a table-filling algorithm that takes advantage of storing intermediate results $\frac{\partial u^{(n)}}{\partial u^{(i)}}$.

- Each node in the graph has a corresponding slot in a table to store the gradient for that node. By filling in these table entries in order, back-propagation avoids repeating many common subexpressions.

- This table-filling strategy is sometimes called dynamic programming.

Example: back-propagation for MLP training

- As an example, we walk through the back-propagation algorithm as it is used to train a multilayer perceptron.
- Here we develop a very simple multilayer perceptron with a single hidden layer.
- To train this model, we will use minibatch stochastic gradient descent.
- The back-propagation algorithm is used to compute the gradient of the cost on a single minibatch.

Example: back-propagation for MLP training

- Specifically, we use a minibatch of examples from the training set formatted as a design matrix $\boldsymbol{X}$ and a vector of associated class labels $\boldsymbol{y}$.

- The network computes a layer of hidden features $\boldsymbol{H} = \max\{0, \boldsymbol{X}\boldsymbol{W}^{(1)}\}$. To simplify the presentation we do not use biases in this model. We assume that our graph language includes a relu operation that can compute $\max\{0, \boldsymbol{Z}\}$ element-wise.

- The predictions of the unnormalized log probabilities over classes are then given by $\boldsymbol{H}\boldsymbol{W}^{(2)}$.

Example: back-propagation for MLP training

- We assume that our graph language includes a $\mathrm{cross\_entropy}$ operation that computes the cross-entropy between the targets y and the probability distribution defined by these unnormalized log probabilities. The resulting cross-entropy defines the cost J_{MLE}. Minimizing this cross-entropy performs maximum likelihood estimation of the classifier.

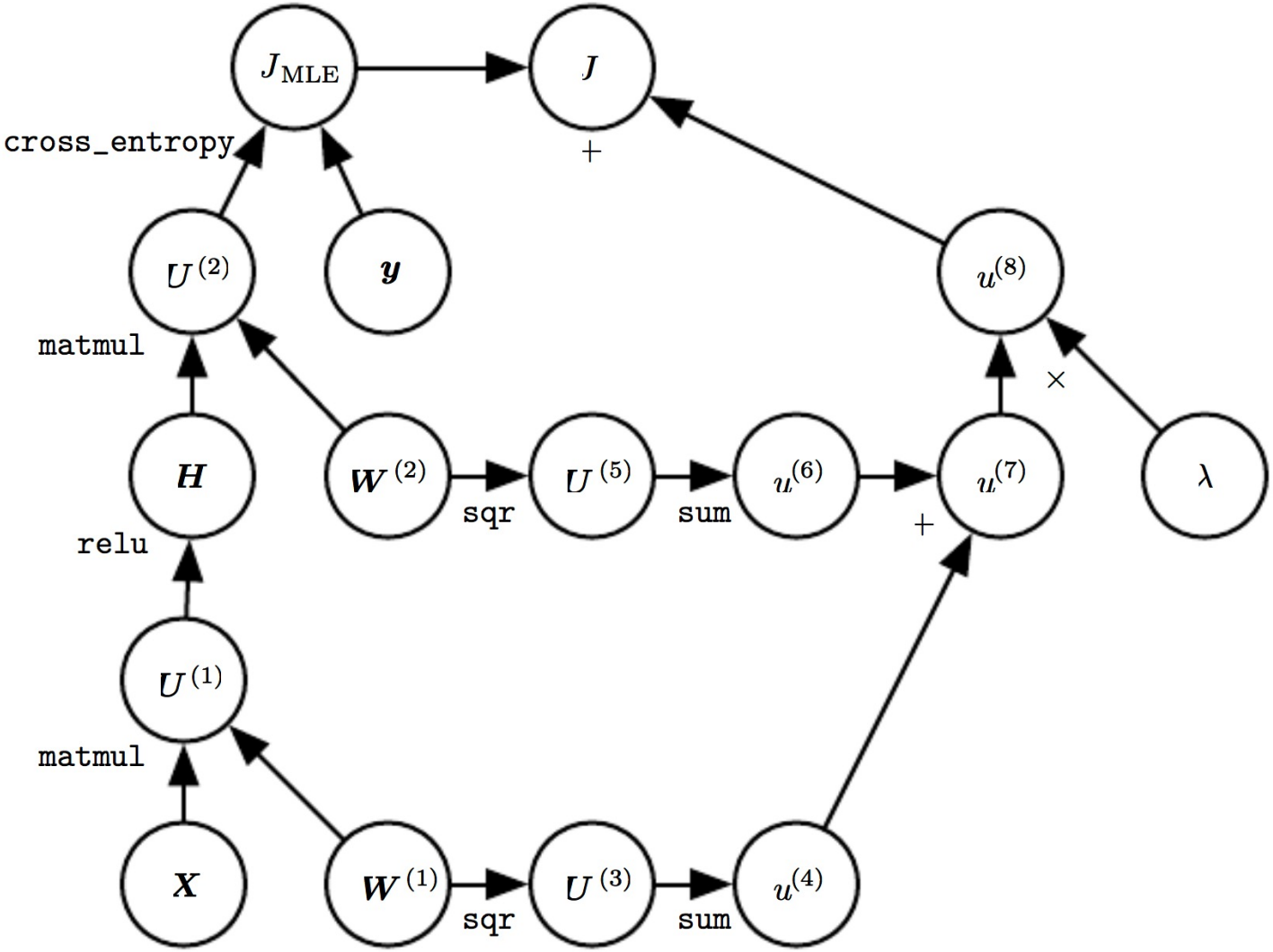
- We also include a regularization term. The total cost

$$J = J_{\mathrm{MLE}} + \lambda \left(\sum_{i,j} \left(W_{i,j}^{(1)} \right)^2 + \sum_{i,j} \left(W_{i,j}^{(2)} \right)^2 \right) \tag{6.56}$$

  consists of the cross-entropy and a weight decay term with coefficient λ.

Example: back-propagation for MLP training

The computational graph for the gradient of this example



Example: back-propagation for MLP training

- To train, we wish to compute both $\nabla_{\boldsymbol{W}^{(1)}} J$ and $\nabla_{\boldsymbol{W}^{(2)}} J$. There are two different paths leading backward from J to the weights: the cross-entropy and the weight decay.

- The weight decay cost will always contribute $2\lambda \boldsymbol{W}^{(i)}$ to the gradient on $\boldsymbol{W}^{(i)}$.

- The other path through the cross-entropy cost is slightly more complicated. Let $\boldsymbol{G}$ be the gradient on the unnormalized log probabilities $\boldsymbol{U}^{(2)}$ provided by the $\mathrm{cross\_entropy}$ operation. The back-propagation algorithm now needs to explore two different branches.

- On the shorter branch, it adds $\boldsymbol{H}^{\mathrm{T}}\boldsymbol{G}$ to the gradient on $\boldsymbol{W}^{(2)}$, using the back-propagation rule for the second argument to the matrix multiplication operation.

Example: back-propagation for MLP training

- The other branch corresponds to the longer chain descending further along the network. First, the back-propagation algorithm computes $\nabla_{\boldsymbol{H}} J = \boldsymbol{G}\boldsymbol{W}^{(2)}$ using the back-propagation rule for the first argument to the matrix multiplication operation.

- Next, the relu operation uses its back-propagation rule to zero out components of the gradient corresponding to entries of $\boldsymbol{U}^{(1)}$ that are less than 0. Let the result be called $\boldsymbol{G}'$.

- The last step of the back-propagation algorithm is to use the back-propagation rule for the second argument of the matmul operation to add $\boldsymbol{X}^{\mathrm{T}}\boldsymbol{G}'$ to the gradient on $\boldsymbol{W}^{(1)}$.

Example: back-propagation for MLP training

- After these gradients have been computed, the gradient descent algorithm, or another optimization algorithm, uses these gradients to update the parameters.

- For the MLP, the computational cost is dominated by the cost of matrix multiplication. During the forward propagation stage, we multiply by each weight matrix, resulting in $O(\omega)$ multiply-adds, where ω is the number of weights.

- During the backward propagation stage, we multiply by the transpose of each weight matrix, which has the same computational cost.

Example: back-propagation for MLP training

- The main memory cost of the algorithm is that we need to store the input to the nonlinearity of the hidden layer. This value is stored from the time it is computed until the backward pass has returned to the same point.

- The memory cost is thus $O(mn_h)$, where m is the number of examples in the minibatch and n_h is the number of hidden units.

Complications

- Our description of the back-propagation algorithm here is simpler than the implementations actually used in practice.

- As noted above, we have restricted the definition of an operation to be a function that returns a single tensor. Most software implementations need to support operations that can return more than one tensor.

- For example, if we wish to compute both the maximum value in a tensor and the index of that value, it is best to compute both in a single pass through memory, so it is most efficient to implement this procedure as a single operation with two outputs.

Complications

- Back-propagation often involves summation of many tensors together. In the naive approach, each of these tensors would be computed separately, then all of them would be added in a second step.

- The naive approach has an overly high memory bottleneck that can be avoided by maintaining a single buffer and adding each value to that buffer as it is computed.

Complications

- Real-world implementations of back-propagation also need to handle various data types, such as 32-bit floating point, 64-bit floating point, and integer values. The policy for handling each of these types takes special care to design.

- Some operations have undefined gradients, and it is important to track these cases and determine whether the gradient requested by the user is undefined.

- Computing higher-order derivatives requires efficient methods (see the textbook).